Written Assignment 6

Introduction To Computer Systems (Carnegie Mellon University)

15-213: Introduction to Computer Systems
Written Assignment 6

This written homework covers code optimization and linking.

Directions

Complete the question(s) on the following pages with single paragraph answers. These questions are not meant to be particularly long! Once you are done, submit this assignment on Canvas.

Below is an example question and answer.

Q: Please describe the benefit of 2s-complement signed integers versus other approaches (such as 1s-complement or signed-magnitude).

A: For both 1s-complement and signed-magnitude representations of signed integers, we end up representing both -0 and +0, which gets inconvenient when the computer wants to test for a zero result. Additionally, in both of these representations, implementing addition/subtraction is complicated. With 2s-complement, the hardware for addition / subtraction is the same for both signed and unsigned inputs.

Grading

Each assignment will be graded in two parts:

1. Does this work indicate any effort? (e.g. it's not copied from a homework for another class or from the book)
2. Three peers will provide short, constructive feedback.

Due Date

This assignment is due on October 21, 11:59 PM EST. Remember to convert this time to the timezone you currently reside in.

Question 1

Compilers are not just tools that can turn source code into executables, they can also make optimizations for the programmer to improve performance. Name one technique (among the many) that compilers use to make optimizations, and identify one guarantee that compilers must deliver when attempting to implement those optimizations.

Question 2

When building a C program, the C source is often first compiled into an assembly file, which is then assembled into an object file before being linked into the final executable. Josh Z is writing a hello world program named **wheat.c** that prints out one of his favorite quotes. The C source is given below.

```
#include "stdio.h"

int main(void) {
  puts("If I am worth anything later,\n"
      "I am worth something now.\n"
      "For wheat is wheat,\n"
      "even if people think it is a grass in the beginning.\n");
  puts("- Vincent Van Gogh\n");
  return 0;
}
```

He compiles this program into assembly with **gcc -S wheat.c**, producing **wheat.s**, then assembles it into a binary object file using **as -o wheat.o wheat.s**. Before building this object file into an executable using **gcc -o wheat wheat.o**, Josh Z pauses to inspect the **wheat.o** object file using his favorite command line tool, objdump.

Josh Z first checks that **wheat.o** contains the correct assembly code by calling **objdump -d wheat.o**, yielding the following output:

```
wheat.o:   file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov   %rsp,%rbp
   4:   bf 00 00 00 00          mov   $0x0,%edi
   9:   e8 00 00 00 00          callq  e <main+0xe>
   e:   bf 00 00 00 00          mov   $0x0,%edi
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   b8 00 00 00 00          mov   $0x0,%eax
  1d:   5d                      pop   %rbp
  1e:   c3                      retq
```

To his surprise, he finds that the addresses of the **`callq`** instructions are nonsensical — they point to within the **`main`** function itself!

Help Josh Z make sense of his objdump results. Why are the addresses of the callq instructions (calling the function **`puts`**) different, and why do they point to instructions inside the **`main`** function? Describe how this strangeness is resolved in the final executable.

Question 3

Grace wrote a vector library in C, then wrote test cases for her library in a `.c` file. She then compiled the library and test cases into an executable, and found one bug in her implementation. When she fixed the vector library, and recompiled, she noticed the recompilation took significantly less time. Why would this be so?